A Modular Approach to Performance, Portability and Productivity for 3D Wave Models

WOLFHPC Workshop

Larisa Stoltzfus
School of Informatics
University of Edinburgh
larisa.stoltzfus@ed.ac.uk

Christophe Dubach
School of Informatics
University of Edinburgh

Michel Steuwer
School of Computing Science
University of Glasgow

Alan Gray
Edinburgh Parallel Computing Centre

Stefan Bilbao
School of Music
University of Edinburgh

ABSTRACT

The HPC hardware landscape is growing increasingly complex in order to meet demands in scientific computing for greater performance. In recent years there has been an explosion of parallel devices coming on to the scene: GPUs, Xeon Phis and FPGAs to name but a few examples. As of writing, even the current leading supercomputer, Sunway TianhuLight, uses its own bespoke on-chip accelerators[12]. Available programming models, however, lag behind and are not currently able to provide the necessary tools for running scientific codes across platforms in ways that are performant, portable and productive.

This environment creates a plethora of challenges for computational scientists of which we focus on two: first the need for a high level of productivity for codes that still get good performance and second consistently getting good performance across platforms - the "performance portability" problem. Existing solutions tend to be either not productive but provide good performance or focus on high-level abstractions requiring heuristics to get good performance (often which are tied to particular platforms). While some current approaches raise the productivity level, they are often trying to solve the same problems over and over or trying to solve too many issues for a niche domain. In addition, many of these approaches have only been tested on simplistic benchmarks, which can lose critical functionality of real-world simulation codes. We instead propose a modular approach using existing frameworks to target these issues separately: a high-level DSL to target the productivity problem compiling into an IR language which addresses the performance portability problem.

Our previous research has shown that the development of more productive and performance portable codes for room acoustics simulations is possible. Preliminary results using the intermediary parallel language LIFT[16] confirm that this framework is capable of handling complex stencils. Further developing LIFT and targeting existing stencil-focused DSLs will create a simple, modularized approach which harnesses and expands existing functionality instead of trying to reinvent the wheel. This modular approach can then be used as an example to extend to other physical simulations using similar algorithms.

WOLFHPC, November 2017, Denver, USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

KEYWORDS

Code generation, heterogenous computing, DSLs, stencils

ACM Reference Format:
Larisa Stoltzfus, Christophe Dubach, Michel Steuwer, Alan Gray, and Stefan Bilbao. 2017. A Modular Approach to Performance, Portability and Productivity for 3D Wave Models. In *Proceedings of WOLFHPC, Denver, USA, November 2017,* 9 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Computer simulations are a critically important tool in scientific fields that bridge theory with reality. Many of these simulations - such as 3D wave-based models like room acoustics[23] or ground penetrating radar[7] - are complicated to model and even more difficult to abstract about. The difficulty in writing abstractions for these algorithms stems from absorbing boundary conditions, multiple timesteps and varying sized stencils used in the models. However, such simulations are integral to predicting the properties and behavior of the physical world around us. As such, the ability to program these simulations in a productive way which can perform well across the increasing range of HPC architectures is of growing importance.

Computing systems have moved towards parallel (as well as heterogeneous) architectures as greater performance can no longer be achieved through a single core. The types of architectures available are changing and increasing in numbers to meet demands for performance for scientific codes which have traditionally been run on CPUs. It is therefore crucial to accommodate portability across both traditional and emerging platforms in order to avoid having to rewrite codes as new platforms emerge. Currently, many scientific groups utilize multiple code bases, which is error-prone and time-consuming to maintain. Computational scientists should be able to focus on their own research and not require HPC expertise for re-tuning and rewriting when newer, more performant platforms emerge. Furthermore, even where codes can be ported from one architecture to another, there is often no guarantee they will retain the same performance level on the new platform.

This paper proposes a novel, modular approach to tackling these issues of performance, portability and productivity for wave simulation models by adapting and connecting existing high-level frameworks. Our application area of interest is 3D wave models

with absorbing boundary conditions. Our approach is two-fold: extend an existing stencil DSL with a high level of productivity to compile into the LIFT language and then enable the LIFT language to accommodate more complex stencils (for 3D wave models) to create performance portable, hardware-agnostic code. The LIFT language acts as an "assembly language of HPC," which can be used for a variety of applications including stencils. It is a pattern-based language and compiler addressing the performance portability problem through automatic exploration of optimizations of rewrite rules[16]. We propose to further develop this language to enable computation and optimization for stencils from 3D wave models. We will then extend a productive top-level DSL to compile into Lift, which can then use Lift's composability and rewrite rules to compile simulations down into performant and portable code. This will create a modular workflow that will enable performance, portability and productivity for 3D wave models which could serve as a template for other types of algorithms.

A large number of high-level parallel methodologies currently exist which focus on stencil applications, however none provide all three of performance, portability and productivity. The strengths and weaknesses of these approaches will be discussed further in Section 2. Then, some background about 3D wave models and previous related work is discussed in Section 3. The results in this section provide baselines (as well as motivation) for our current work with LIFT. Next, the LIFT intermediary language which addresses the performance portability challenge is introduced in Section 4. Current and ongoing work is discussed at length in Section 5. Future work involving the LIFT framework is covered in Section 6. Finally, the paper concludes in Section 7.

2 PERFORMANCE AND PRODUCTIVITY IN EXISTING APPROACHES FOR STENCIL COMPUTATIONS

There are currently a wide range of potential frameworks and tools available that aim to provide higher levels of productivity, portability, performance and combinations therein which could accommodate stencil computations like 3D wave models. At the low-level end (or libraries and tools that are much closer to machine level) there exist a number of libraries or language enhancements. However, these low-level solutions do not provide productivity, so we will focus on high level solutions because this characteristic is crucial. Higher-level abstraction solutions include skeleton frameworks, code generators, DSLs and others. These high-level approaches focus more on distinct layers of abstraction that are far removed from the original codes and which generally aim to support a higher level of productivity for the programmer. Many of these higher level approaches even focus in particular on stencil applications (ie. Halide[15], Pochoir[22] and Exastencils[11]).

2.1 Algorithmic Skeletons

Skeleton frameworks are a large subgroup of abstraction solutions developed for enhancing productivity. These skeletons focus on the idea that many parallel algorithms can be broken down into pre-defined building blocks[4]. Thus, an existing code could be embedded into a skeleton framework that already has an abstraction and API built for that algorithm type, such as a stencil. These frameworks then simplify the process of writing complex parallel code (like OpenCL) by providing an interface which masks the low-level syntax and boilerplate. A number of these skeleton frameworks have been developed, many of which are intended for grid-based applications, however few have been tested on larger simulation models. Additionally, many of them lack 3D functionality, such as Skepu[6] and SkelCL[17] which only support 1D and 2D stencils. Some of the libraries also only target particular architectures - for example PSkel, which does support 3D stencils, but only ports to NVIDIA GPUs[14].

2.2 Code Generators

Code generators translate or compile source code from one language into another. They are a promising area in this field of research given that their modularity allows for flexibility between languages and platforms. Petabricks is an example of a code generator, which is actually a language and a compiler capable of autotuning over multiple pre-existing implementations of algorithms to tailor to a specific hardware[1]. Kokkos and SYCL are two other code generators that are gaining popularity, which compile down to CUDA and OpenCL respectively, and are tailored more towards general use and ease of programmability[5, 10]. Kokkos also has a sophisticated memory mapping pattern for optimizing codes. SYCL focuses more on bringing a simplified interface to OpenCL in C++. None of these offer stencil-specific support for improving productivity, however, and they both miss out on automatic performance portability. There is however, one code generation framework with a similar approach to LIFT: Delite[21] also utilizes the concept of "parallel patterns," however still relies on hard-coded optimizations.

2.3 Domain Specific Languages

There are a large number of DSLs available, many of these focusing on stencils including: Exastencils, Halide, Pochoir and others. However, none of these support stencils with absorbing boundary conditions nor do they achieve full performance portability without heuristics. Exastencils is a DSL developed at the University of Passau that aims to create a layered framework that uses domain specific optimizations to build performant portable stencil applications[11]. Halide is another functional DSL with auto-tuning that specializes in abstracting stencils by separating algorithm from execution[15]. Exastencils aims for performance portability, however its approach is through the use of a tree of heuristics. Halide focuses primarily on image processing stencils, which have less in common with 3D wave models. Pochoir is limited to particular hardware. Ideally these DSLs could focus on abstraction, while another level of code addresses the performance portability problem.

2.4 Limitations of Current Approaches

While these high-level frameworks take fairly different approaches, none are without problems. Skeletons enable ease of programmability and portability, but they alone are not enough to produce performance-portable code without heuristics, especially as they are often tied to a particular framework or architecture. A framework that remains de-coupled from specific implementations would avoid this problem. For the higher-level code generation frameworks,

performance results look promising in comparison to hand-tuned optimizations, though generally only small benchmarks have been tested or broad domains focused on. Some of the stencil-specific frameworks (like Halide) also focus mainly on images, simple stencils or other non-HPC specific domains. The main focus of DSLs also tends to be on productivity, which means that performance and portability often lags behind or codes are only performant for particular architectures. These limitations mean that there currently are no solutions that have been shown to give 3D wave model simulations good performance, portability and productivity.

3 ROOM ACOUSTICS AND OTHER 3D WAVE MODELS

3D wave-based simulations are an important tool in physics for modeling the evolution of waves through space and time of various mediums. Room acoustics simulations are an example of such a simulation and simplified benchmarks of this type of simulation have been studied previously in more depth[20]. Performance and bandwidth were compared across different implementations, as well as different data abstractions of the same implementation using a bespoke framework. This framework, called *abstractCL* was developed for room acoustic simulations to test out different optimizations and data abstractions without rewriting the code. It shows that productivity can be lifted for room codes, while retaining performance across different hardware. The results of this research provide baseline comparison points for later work with the LIFT framework. As well as this, the development of this niche framework motivates the need for a completely different type of approach than just writing a library to work with room codes.

3.1 Overview

The finite difference time domain method (FDTD) is a widely used numerical approach for modelling of the 3D wave equation[3], which is shown in Equation 1.

$$\frac{\partial^2 \Psi}{\partial t^2} = c^2 \nabla^2 \Psi \tag{1}$$

Space is discritised into a three-dimensional grid of points, with data values resident at each point representing the acoustic field at that point. The state of the system evolves through time-stepping: values at each point are repeatedly updated using "finite differences" of values in the neighborhood of that point. This algorithm (also known as a stencil) updates points as determined by the choice of discretisation scheme for the partial differential operators in the wave equation [2]. This approach can be computationally expensive, however parallelization techniques have shown great improvements in performance. Though there has been progress in the development of techniques to exploit modern parallel hardware, much of it is low-level or tied to specific platforms. Ideally, scientific models should be able to run in a portable manner across different architectures while retaining performance and being straightforward to program.

Room acoustics simulations were developed to model sound waves in an enclosed three dimensional space. These simulations use first physical principals to represent the properties of sound as it moves through space and time. "Room codes" use grids as data types that update values via the commonly used nearest-neighbors technique. The output of these simulations can provide composers or architects the ability to hear what a composition or noise would sound like in a space without actually being there or even having built it. However, there are many other types of wave-based simulations, such as ground penetrating radar or lattice boltzman, which use similar techniques for more accurate modeling. Thus, the results found here specifically for room acoustics codes could readily be applied to other, similar wave based models.

3.2 Previous Work

Our previous research has shown that it is possible to develop more productive simulation codes such as room acoustics, while still achieving high performance across platforms. A study was done across various architectures with a number of different implementations of the same room acoustics benchmark using different programming models to determine limiting factors and behavior. Memory bandwidth was shown to be the limiting factor for this benchmark as can be seen that it runs fastest on the platform with the highest memory bandwidth (AMD R9 295X2). In addition, a more productive, bespoke framework was developed to test out different data layouts and optimizations without rewriting the code.

3.2.1 Performance Comparison. Four implementations of the same room acoustics benchmark (CUDA, OpenCL, abstractCL and targetDP) were run on six platforms (Intel Xeon CPU, Xeon Phi Knights Corner, NVIDIA K20, NVIDIA GTX 780, AMD R280 and AMD R9 259X2). Obviously not all versions were able to run on all platforms - CUDA and targetDP - are limited to NVIDIA platform GPUs. All other versions utilize OpenCL which can run on all selected platforms. *targetDP* is a low-level library for running simulation codes on NVIDIA GPUs, CPUs or Xeon Phis without rewriting the codes[8]. *abstractCL* is the developed framework tied to room acoustics simulations that we built using C++ macros and OpenCL to allow simplified swapping in and out of data layouts and optimizations[20]. Specifications for the various platforms can be seen in Table 1.

Performance results of the runs of the four different versions can be seen in Figure 1. The various colors indicate how much time was spent in which part of the code: red for the time in the room update kernel, orange for the time spent in aggregating values at the receiver point kernel, green for the time spent copying data around and blue for all other miscellaneous time. The graph is split up into six sections, one for each of the platforms run on, which is indicated at the top of each section of the graph. Then, the bars in each section show the performance of the version of the code that has produced the result. Performance was generally comparable on all GPU platforms and more widely pronounced on the Xeon Phi and CPU (though getting reproducible results on the Xeon Phi proved to be tricky). However performance across platforms differed tremendously, up to 40-50% between GPUs.

In addition to comparisons of the original benchmark, comparisons were done with an optimized version of the room acoustics code. More about the 2.5D tiling optimization used in this version can be found in Section 5.2. These comparisons involved the use of local and image memory and through the use of *abstractCL*, this could be swapped in and out easily without having to rewrite the

Platform	Number of Cores/ Stream Processor	Peak Bandwidth (GB/s)	Peak GFlops (Double Precision)	Memory (MB)
NVIDIA K20	2496	208	1175	5120
NVIDIA GTX 780	2304	288.4	165.7	3072
AMD R280	2048	288	870	3072
AMD R9 259X2	2816	320	716.67	4096
Xeon Phi 5110P	60	320	1011	8000
Intel Xeon E5-2670	32	51.2	166.4	126000

Table 1: Platform Specification Table. This table shows the specifications of the various platforms used in previous work on comparing versions of a room acoustics simulation.

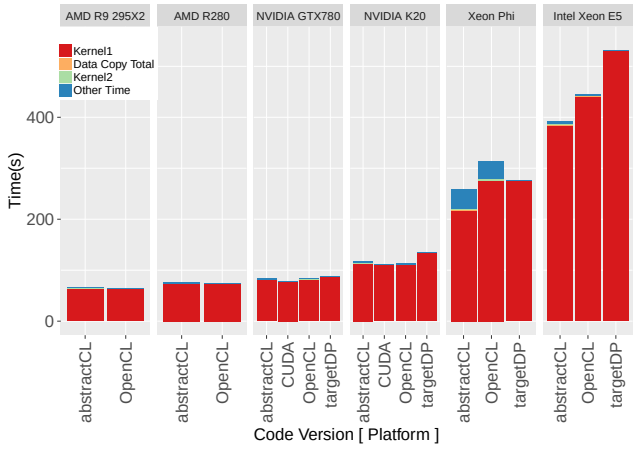

Figure 1: Performance timings for simple room acoustics benchmark implementations across various platforms using room sizes of 512x512x404 grid points.

whole benchmark. Results of adding these optimizations can be seen in Figure 2, where the blue graphs show more optimal memory usage than the original version shown in green which led to performance increases of up to 15%.

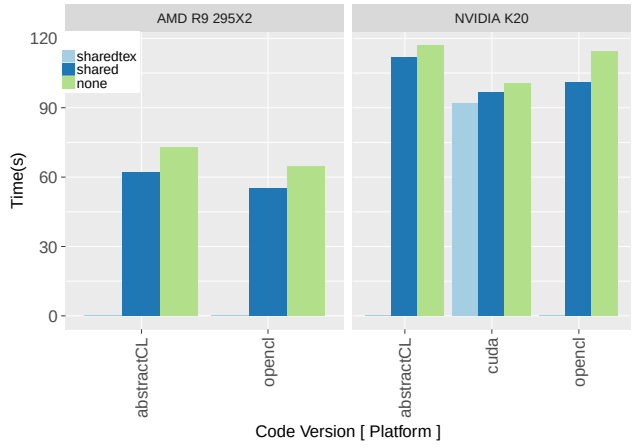

Figure 2: Local memory optimizations for CUDA, OpenCL and abstractCL versions of the room acoustics benchmarks run for room sizes of 512x512x404 grid points.

3.2.2 abstractCL. A new framework, *abstractCL*, was created as a more productive solution for generating room simulation kernels on-the-fly, depending on the type of run a user wants to do. The type of variations can be between different data layouts of the grid passed in to represent the room, hardware-specific optimizations or both. This is done through flags which swap in and out relevant files that include overloaded functions and definitions in the main algorithm itself. Certain functions must always be defined as dictated by a parent class. Those functions' implementations can be pulled in from different sources and concatenated together to create the simulation kernel before compilation. The data abstractions and optimizations investigated for this project include: thread configuration settings, memory layouts and memory optimizations. Algorithmic changes can be introduced by adding new classes to the current template for more complicated codes.

This framework runs similarly to the other benchmark versions, apart from that the kernel is created before the code is run (which creates some initial overhead). It was developed in C++ (due its built-in functionality for classes, templates, inheritance and strings) as well as OpenCL. However, as shown in Figure 1, the framework produces code that is on par with hand-tuned OpenCL codes. However it raises the level of productivity, which in previous work we have shown comparisons of to other versions using LOC (lines of code), where *abstractCL* has significantly fewer[19].

Optimizations must be tuned manually for each architecture, which is a limitation in its performance portability. Another limitation of this framework is that while it could be extended to other types of models, it is currently only designed for room acoustic simulations. Additionally, it is still relatively low-level compared to other DSLs. Finally, though it can provide some level of performance portability, this is done through hard-coded optimizations. Our goal is to improve upon all of these issues with our modularized approach using the LIFT language.

4 A MODULAR APPROACH USING THE LIFT LANGUAGE

In the next section, we discuss our modular approach aimed at achieving productivity, performance and portability automatically using the LIFT language. However, LIFT is more than just a language: it is also a framework. The language provides the algorithmic primitives to compose algorithms in and the framework handles the language compilation, rewrite rule search space and low-level code optimization. Its goal is to target a multitude of application domains, not just stencils, so the bespoke language used in the framework is rich in functionality. To better understand this process, we step through a simplified example of a room acoustic simulation written in LIFT.

4.1 Our Vision

Instead of writing yet another stencil library or DSL to address the shortcomings of current solutions, this project aims to take advantage of what is available already and create a modularized approach to HPC 3D wave model development using existing DSLs and the LIFT framework. In an ideal world, DSLs would use a common compiler, so that the writers of these languages could focus on the needs of their specific domain in the abstraction layer instead of

also having to manage hardware-specific low-level optimizations. This is where the LIFT language provides great leverage. LIFT is designed to be used as an intermediate layer targeted by DSLs to handle the low-level implementation details and relevant optimizations. Unlike many existing approaches, the LIFT language and its corresponding compiler are designed to address the performance portability challenge as a first priority. Because of its modular design using a suite of composable algorithmic primitives, LIFT can be used to target a host of different applications. This also means that LIFT is hardware-agnostic and can readily be adopted to future platforms.

4.2 Lift Overview

The LIFT language has been developed in Scala as a modular solution to the performance portability problem[16, 18]. The idea is that by separating HPC programming into separate levels of abstraction, then each level can be restricted to its particular purpose. An overview of this process can be seen in Figure 3. At the top level ("high level abstractions") a DSL would compile into the LIFT language which is comprised of a suite of algorithmic primitives (such as *map*, *reduce*, *zip*, etc). This may require a version of an API to be developed in LIFT to connect these layers more fluidly. Different versions of an algorithm composed using these primitives (as well as more low level language specific ones) can then be explored using rewrite rules, which explore different optimization options. These versions can then be run on different platforms to determine the best-performing version for a different platform and this is done by generating OpenCL kernels from the different rewrite rule versions. The only backend that LIFT currently supports is OpenCL, which provides portability to a number of different hardware platforms.

The LIFT language provides three key functionalities: (1) functional, reusable high-level primitives (2) rewrite rules describing exchangeable relationships between compositions of primitives and (3) a code generating search space which determines the best version of a kernel to run on a particular platform. At the higher level, the language is composed of "reusable primitives," written in a functional style. Algorithms for a wide variety of applications (including stencils) can be rewritten as compositions of these primitives. "Rewrite rules" describe a variety of transformations of a given algorithm decomposition, which are formalized and proven not to change a program's semantics. While the only backend LIFT supports so far is OpenCL, the modular design of the language makes it easily extensible to other parallel programming frameworks. One limitation of LIFT is that it is not productive on its own, however we see this as a feature meaning the language can focus on producing portable and performant code for a wide range of applications without having to provide a high-level interface to program them in.

4.3 Working Example: Room Acoustics Simulation in Lift

In order to better explain how LIFT works, we step through a simplified example of a room acoustics simulation benchmark. This algorithm can be seen in Listing 1. This simulation was developed by HPC physicists [23] and models the behavior of a sound wave as it propagates from a source to a receiver in an enclosed 3-dimensional

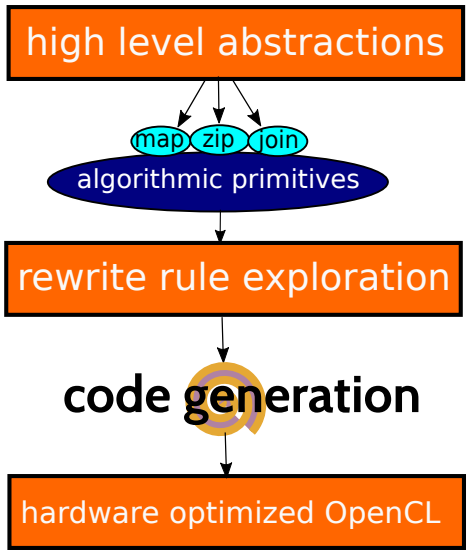

Figure 3: Overview of the LIFT Framework

space (the updates at the receiver end are done in another less computationally intensive kernel not shown here). Where the sound encounters a physical boundary (in this example, the walls are only the boundaries of the grid), the coefficients used in calculating the physical properties of the sound wave are adjusted according to the reflection. Sound waves are discritised in space as well as time, but only spatial discretisation is parallelizable, which is shown in this example.

Multiple Grid Inputs The two inputs used in this benchmark ($grid_{t-1}$ and $grid_t$ on lines 1- 2) are the same size and indicate previous and current time steps in order to update the state of the room across time (and which get swapped on each iteration). This type of inputs is often found in real world physical simulations, which span three dimensions for physical space and one for time. These grids are zipped together as one input to the algorithm along with an on-the-fly array which calculates the number of neighbors for a given point based on a provided function (`computeNumNeighbors`). This last grid serves as a mask for the boundaries. The first grid is taken point-by-point, however the second grid uses *slide3D* to calculate 27-point neighborhoods around the point of interest in order to retain neighboring points for the stencil. In this 3D case, *slide* returns a cube of values each one point away from the original value resulting in the 27 points. The number of neighborhoods correctly matches up to the size of the $grid_t$ input array as the $grid_{t-1}$ input is padded using the *pad3D* primitive first so that no out of bounds accesses occur. These inputs are then zipped together with their number of neighbors (same for each grid) resulting in a tuple of: $\{\text{VALUE}_{t-1}, \text{NEIGHBORHOOD}_t, \text{NUMNEIGHBORS}\}$ as seen on lines 14- 17. The resulting output of the next timestep of the room state is calculated using parts of all three elements of this tuple.

Calculating the Stencil For the neighborhood part of the amalgamated input, the *at* primitive (expressed with []) makes it easy to calculate a more complicated stencil as can be seen on lines 5- 7. The result can then be combined with the other inputs in an equation

```
1  acousticStencil(grid_{t-1}:[[[float]_m]_n]_o,
2                  grid_t:[[[float]_m]_n]_o) {
3    map3D(m -> {
4     val valueGrid_t = m.0
5     val sumGrid_{t-1} =
6      m.1[0][1][1] + m.1[1][0][1] + m.1[1][1][0] +
7      m.1[1][1][2] + m.1[1][2][1] + m.1[2][1][1]
8      val numNeighbor = m.2
9      return getCF(m.2, CSTloss1, 1.0f) * ((2.0f -
10     CST_l2 * numNeighbor) * m.1[1][1][1] +
11     CST_l2 * sumGrid_{t-1} - getCF(m.2,
12     CSTloss2, 1.0f) * valueGrid_t)
13   },
14   zip3D(grid_t,
15         slide3D(3, 1,
16                 pad3D(1,1,1,zero,grid_{t-1})),
17         boundaryGrid(m,n,o,computeNumNeighbors))
               )
18 }
```

Listing 1: Room acoustic simulation as expressed in the LIFT language

to model the sound (lines 9–12). In this case, 27 points are passed in for the neighborhood of points one value away in any 3D direction and any of these points could be pulled out in any shape. In this instance, only 7 points are used for the equation to calculate the stencil: values to the left, right, up, down, top and bottom.

Boundary Handling One of the most difficult problems for wave-based simulations is accurate physical boundary handling, which can involve the use of states to retain memory. This simplified version uses state-free boundary conditions, which involve variable coefficients, however the same ideas could be applied to more complicated state conditions. The variable coefficients (also known as "loss" at the boundary, in the physical sense) are handled through the use of a mask, which returns a different value depending on whether it is on a boundary or not. The mask is calculated on the fly as an input using the *boundaryGrid* generator function and contains a value at each point in the grid of the number of available neighbors for a point (ranging from 3 at a corner to 6 on the insides). The coefficients are then calculated using the *getCF* function as can be seen on line 9. For those values which are on the border (i.e., *numNeighbors* < 6), a lossy coefficient is used in the equation (*CSTloss1* or *CSTloss2*). The overall computation is based on a discretized version of the 3D wave equation to simulate the energy at different points in the room.

5 3D WAVE MODEL DEVELOPMENT USING THE LIFT FRAMEWORK

Our current work focuses on enhancing LIFT to accommodate room acoustics stencils. These codes now run with comparable results to the original hand-written versions (those shown in Section 3). Several additions to the language have been made to obtain these results. Performance still lags behind optimized versions, however, so 2.5D-tiling (which was used in versions of the original benchmark) and other optimizations are being investigated as additions to the language as well.

5.1 Updating Lift to Accommodate Complex 3D Stencils

Preliminary results have shown that LIFT is capable of expressing stencils of varying types and sizes[9]. In particular, simplified room acoustics simulations have been thoroughly investigated in the framework and a number of other 2D and 3D stencil benchmarks have also been implemented. Additionally, ground penetrating radar algorithms are also being implemented in the language as they have similar wave behavior to room acoustics. Though both simulations are modeled using the 3D wave equation and have absorbing boundary conditions, GPR codes require modeling both the electric and magnetic fields interacting with each other.

Performance values for various versions of room acoustics implementations can be seen in Figure 4. The two bars furthest to the right on each of the platforms show the original room acoustics codes written in C and OpenCL. The black line across the graph indicates the level of the unoptimized original benchmark. The three bars on the left are versions of the LIFT implementations with optimization additions. As can be seen, these have dramatically improved the implementation, but have not yet completely closed the performance gap of the optimized version of the original code.

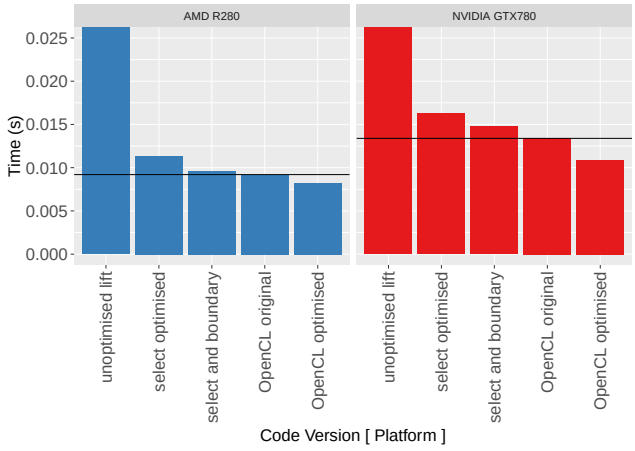

Figure 4: Comparison of optimization versions in the LIFT framework implementation of room acoustic benchmark with grid size of 512x512x404 grid points.

As shown, implementation of the room acoustic stencils involved two stages of optimizations. These were adding a *select* and a *boundary* optimization. Figure 5 shows a visual representation of this as well as how the code differs in C versus LIFT. At the top left, it can be seen that the neighborhoods produced are pulled out into the shape of interest - in this case a 5-point 2D stencil - using *select*. Previously, all nine values in memory were being accessed as that is what the neighborhood collected by *slide* returns. This is shown by the blue square blotted out by a green cross, showing that only the green cross values are now being accessed. Additionally, masks of the same size as input arrays were originally passed in as parameters in order to determine where boundaries lay. The *boundary* optimization now computes these values on-the-fly, again leading

to fewer unnecessary computations. This can be seen below left in Figure 5, where an on-the-fly mask is now used in conjunction with the original array instead of passing in a hard-coded mask. This depiction shows that originally matrices of values of the room were having to be pre-padded manually to show where the boundary was. Now, these grids can be padded on-the-fly without passing in extra data.

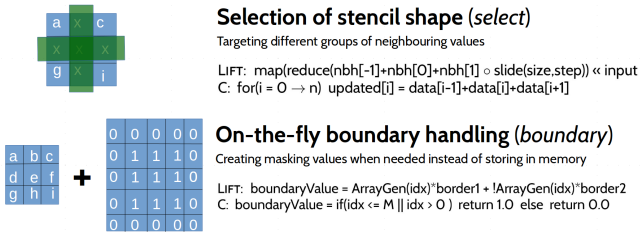

Figure 5: Visual representation of the *select* and *boundary* optimizations in LIFT.

5.2 Optimizations for Rewrite Rules for Stencils

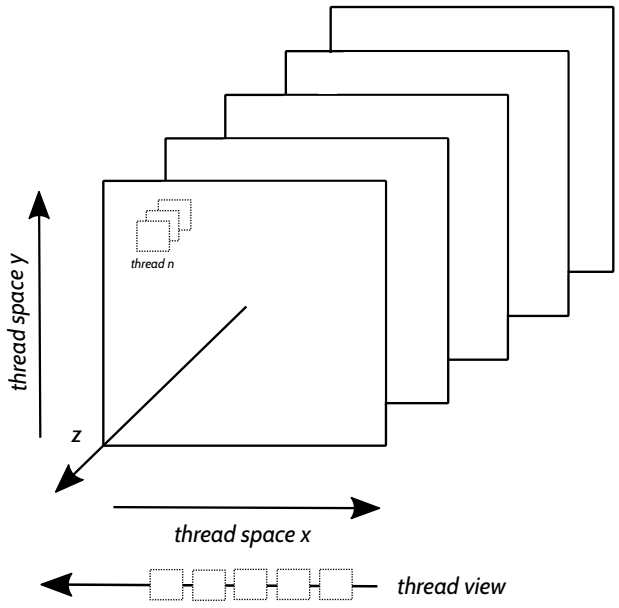

Figure 6: A visual representation of the 2.5D tiling optimization for 3D stencil codes.

The performance of 3D stencils, however, still lags behind optimized versions of the original benchmark, so current work is focusing on developing and formalizing stencil optimizations for 3D models, beginning with 2.5D tiling[13]. The intention is for this optimization (and any others added) to be encoded as a rewrite rule when it is finished. This is because it does not necessarily give better performance for all architectures and grid sizes. Thus, as a rewrite rule, it can be tested out for more optimal performance and be rejected where it does not provide that.

For the 2.5D tiling optimization, previous results showed performance improvements of up to 15% when using this method with local memory on GPUs for room acoustics benchmarks[19]. Figure 2 in Section 3.2.1 shows these results. However, this optimization has been used elsewhere successfully[13] - it is not just limited to room acoustics simulations but can be used with any 3D time-stepping stencil. This method can be described as an XY-tiling method that iterates sequentially over the Z dimension of the room. This means that the Z index of a grid point is held constant while X and Y indices are calculated for a tile spanning the XY plane. For subsequent iterations, the Z index is incremented and the next tile is updated. Local memory is also utilized for points which are reused across tiles. This method is shown visually in Figure 6. The large tiles represent the L number of 2D tiles that comprise a $JxKxL$ sized 3D grid. The smaller internal tiles represent the smaller grid that each thread computes over. The thread space is then divided up into two dimensions instead of three, where the third dimension (L) is computed over sequentially.

To add this method in LIFT, the optimization needs to be decomposed into a LIFT language primitive that can be written in functional ways. We have done this by implementing a *mapseq* followed by a *slide*. *Mapseq* stands for "map sequential." Whereas normally a map can apply a function in parallel, a *mapseq* performs a function across a dimension sequentially. This means that the sequential loop for the Z-dimension is formed by the *mapseq* and the neighborhoods to access the stencil (and reuse memory) are created by the *slide*. How these primitives work independently is described in more detail in [18].

This work has proven challenging as many sequential algorithms do not have a natural one-to-one mapping in functional languages. Overcoming this has required several iterations of brainstorm, test, throw away, repeat. While we currently have a solution available for simple 3D (and similarly 2D and 1D) stencils such as Jacobi, implementing this optimization specifically for room acoustics codes (or similar "time-stepping" stencils) requires more work. This is because the values in the stencil must correspond to previous values in the previous timestep which complicates the windows created for memory accesses. We are still working on overcoming this issue by enabling shapes to be imprinted using this primitive, in the same fashion as the shape optimization as explained in Section 5.1.

6 FUTURE WORK

So far only simplified versions of 3D wave models have been implemented in LIFT. Thus, how to best abstract out absorbing boundary conditions needs to be investigated in more detail and primitives to accommodate these conditions need to be designed and added. Additionally, a stencil-based DSL needs to be extended to compile into the LIFT language for use as a front end. As well as these larger contributions, smaller issues such as targeting multiple cores or GPUs and iterative stencils need to be explored further.

6.1 Absorbing Boundary Conditions

So far the room acoustics benchmarks used in this project have been "state-free" or in other words using constant values at the boundary.

While this produces a usable model, for better accuracy states need to be maintained at the boundary to model the absorption of waves there. These need to be retained for all of the timesteps being used (so two in the current benchmark, but at least three in more advanced codes) and updated accordingly for each iteration. This is difficult to model both because it makes the algorithm more computationally intense, but also makes memory accesses for the states non-contiguous.

6.2 DSL Extension

Lift is not a productive language intended for use by computational scientists. Though it does have "high-level" primitives, these are not intended to be programmed directly. Instead, lift is an intermediate language: an algorithm should be programmed in a higher level DSL that compiles into lift, which then handles the generation efficient of hardware-optimized code. Of the DSLs investigated, Exastencils seems the best potential fit so far for extending to 3D wave models and compiling into lift, as it already uses a modular approach. First it will need to be extended to handle these advanced boundary conditions. Then, it will need to be adapted to compile into the lift language. lift may also require a sort of API to communicate with the DSL in a layer above. The lift language already handles the low-level code generation. Connecting this pipeline of functionality in a modular way allows for each layer to focus on its own part.

6.3 Updating Lift

Currently lift is missing a few integral features that would allow for large scale simulations to run. First of all, iterative stencils are not a natural fit to the language. This is crucial for time-stepping models like room acoustics and ground-penetrating radar which swap arrays of values corresponding to snapshots in time at each iteration. Secondly, many large scale simulations require the ability to program across multiple devices. This is not something that lift currently supports, but would be necessary for widespread adoption. Additionally it would be ideal for lift to accommodate more backends. Currently, it only supports OpenCL, which while portable to many devices is not a universal answer.

7 CONCLUSION

The goal of this project is to provide a modular, reusable workflow for developing performance portable and easily programmable physical simulation models, in particular 3D wave models with absorbing boundary conditions. Developing this will largely involve three steps: extending the lift intermediary language to support complex 3D stencils, extending an existing stencil-focused DSL to support these stencils as well and then adding functionality for this DSL to compile into the lift language. In this manner, we will combine the strengths of the optimizing code generator lift - addressing the performance portability problem - with a stencil-focused DSL, offering good productivity. Other types of DSLs could then follow a similar approach, given that lift is composed of reusable primitives intended to be used with a variety of different applications. Ideally, this will lead by example to other opportunities for performance portable and programmable code for scientific models targeting HPC systems.

8 ACKNOWLEDGMENTS

This work was supported in part by the EPSRC Centre for Doctoral Training in Pervasive Parallelism, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L01503X/1) and the University of Edinburgh.

REFERENCES

[1] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. *PetaBricks: A Language and Compiler for Algorithmic Choice*. Vol. 44. ACM.

[2] Stefan Bilbao, Brian Hamilton, Alberto Torin, et al. 2013. Large Scale Physical Modeling Sound Synthesis. In *Stockholm Musical Acoustics Conference (SMAC)*. 593–600.

[3] Dick Botteldooren. 1995. Finite-Difference Time-Domain Simulation Of Low-Frequency Room Acoustic Problems. *The Journal of the Acoustical Society of America* 98, 6 (1995), 3302–3308.

[4] Murray Cole. 1989. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Ph.D. Dissertation. University of Edinburgh. http://homepages.inf.ed.ac.uk/mic/Pubs/skeletonbook.ps.gz

[5] H Carter Edwards, Christian R Trott, and Daniel Sunderland. 2014. Kokkos: Enabling Manycore Performance Portability Through Polymorphic Memory Access Patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216.

[6] Johan Enmyren and Christoph Kessler. 2010. SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems. HLPP 2010: Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications, Baltimore, Maryland. https://www.ida.liu.se/~chrke55/skepu/SkePU-HLPP-2010.pdf

[7] Antonis Giannopoulos. 2005. Modelling Ground Penetrating Radar By GprMax. *Construction and building materials* 19, 10 (2005), 755–762.

[8] Alan Gray and Kevin Stratford. 2014. targetDP: An Abstraction of Lattice Based Parallelism with Portable Performance. In *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS), 2014 IEEE Intl Conf on*. IEEE, 312–315.

[9] Bastian Hagedorn. 2016. *An Extension of a Functional Intermediate Language for Parallelizing Stencil Computations and Its Optimizing GPU Implementation Using OpenCL*. Master's thesis. University of Muenster. http://www.lift-project.org/publications/2016/hagedorn16masterthesis.pdf

[10] Ronan Keryell, Ruyman Reyes, and Lee Howes. 2015. Khronos SYCL for OpenCL: a Tutorial. In *Proceedings of the 3rd International Workshop on OpenCL*. ACM, 24.

[11] Christian Lengauer, Sven Apel, Matthias Bolten, Armin Größlinger, Frank Hannig, Harald Köstler, Ulrich Rüde, Jürgen Teich, Alexander Grebhahn, Stefan Kronawitter, et al. 2014. Exastencils: Advanced Stencil-Code Engineering. In *European Conference on Parallel Processing*. Springer, 553–564.

[12] Timothy Prickett Morgan. 2016. China's Triple Play for Pre-Exascale Systems. (July 2016). Retrieved August 24, 2017 from https://www.nextplatform.com/2016/07/11/chinas-triple-play-pre-exascale-systems/

[13] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 2010. 3.5-D Blocking Optimization For Stencil Computations On Modern CPUs And GPUs. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*. IEEE, 1–13.

[14] Alyson D Pereira, Luiz Ramos, and Luís FW Góes. 2015. PSkel: A Stencil Programming Framework for CPU-GPU Systems. *Concurrency and Computation: Practice and Experience* 27, 17 (2015), 4938–4953.

[15] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language And Compiler For Optimizing Parallelism, Locality, And Recomputation In Image Processing Pipelines. *ACM SIGPLAN Notices* 48, 6 (2013), 519–530.

[16] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating Performance Portable Code Using Rewrite Rules: From High-Level Functional Expressions To High-Performance OpenCL Code. ICFP 2015 Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, Vancouver, Canada. http://homepages.inf.ed.ac.uk/slindley/papers/array-gpu-draft-february2015.pdf

[17] Michel Steuwer, Michael Haidl, Stefan Breuer, and Sergei Gorlatch. 2014. High-Level Programming of Stencil Computations on Multi-GPU Systems Using the SkelCL Library. *Parallel Processing Letters* 24, 3 (Sept. 2014). http://homepages.inf.ed.ac.uk/msteuwer/papers/ppl2014.pdf

[18] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: A Functional Data-Parallel IR For High-Performance GPU Code Generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. IEEE Press, 74–85.

[19] Larisa Stoltzfus. 2016. *Performance, Portability and Productivity for Room Acoustics Codes*. Master's thesis. University of Edinburgh. http://homepages.inf.ed.ac.uk/s1147290/papers/StoltzfusMasters.pdf

[20] Larisa Stoltzfus, Alan Gray, Christophe Dubach, and Stefan Bilbao. 2017. Performance Portability for Room Acoustics Simulations . *International Conference on Digital Audio Effects* (2017). http://homepages.inf.ed.ac.uk/s1147290/papers/StoltzfusDafx.pdf

[21] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture For Performance-Oriented Embedded Domain-Specific Languages. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 4s (2014), 134.

[22] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. 2011. The Pochoir Stencil Compiler. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 117–128.

[23] Craig Webb. 2014. *Parallel Computation Techniques For Virtual Acoustics And Physical Modelling Synthesis*. Ph.D. Dissertation. University of Edinburgh. www.ness-music.eu/wp-content/uploads/2014/07/CJWebb_thesis-1.pdf